Semester 2, 2018: Lab 4

S2 2018

Lab 4

Note: This lab has more programming problems than we expect everyone to finish during the lab time. If you do not have time to finish all problems, you can continue working on them later (at home, if you have a computer with python set up, in the CSIT lab rooms outside teaching hours, or on one of the computers available across campus), or return to them in a later lab.

Objectives

The purpose of this week's lab is to:

- understand the indexing of (1-dimensional) sequences;
- do some computations over sequences that require iterating through them using loops; and
- practice reading and debugging code.

Exercise 0: Reading and debugging code

The following are attempts to define a function that takes three (numeric) arguments and checks if any one of them is equal to the sum of the other two. For example, `any_one_is_sum(1, 3, 2)` should return `True` (because `3 == 1 + 2`), while `any_one_is_sum(0, 1, 2)` should return `False`.

(a) All of the functions below are incorrect. For each of them, find examples of arguments that cause it to *return* the wrong answer.

Function 1

```
def any_one_is_sum(a,b,c):
    sum_c=a+b
    sum_b=a+c
    sum_a=b+c
```

```
    if sum_c == a+b:
        if sum_b == c+a:
            if sum_a == b+c:
                return True
    else:
        return False
```

Function 2

```
def any_one_is_sum(a,b,c):
    if b + c == a:
        print(True)
    if c + b == a:
        print(True)
    else:
        print(False)
    return False
```

Function 3

```
def any_one_is_sum(a, b, c):
    if a+b==c and a+c==b:
        return True
   else:
        return False
```

(b) For each of the three functions above, can you work out how they are *intended* to work? That is, what was the idea of the programmer who wrote them? What comments would be useful to add to explain the thinking? Is it possible to fix them by making only a small change to each function?

Exercise 1: Debugging loops

The following are two attempts to define a function that computes a sum. (The sum is an approximation of the integral of the function 1/x over the interval from `lower` to `upper`. The bugs in each function below are unrelated to this particular function.) Each function uses a `while` loop that may never terminate. The loop may terminate for some arguments, but not for others.

For each of the functions, find arguments that cause the loop to terminate as well as arguments that cause it to not terminate. The arguments should all be numbers, `lower` should be less than `upper`, and `nterms` should be a positive integer (not zero).

Hint: Add `print` calls *inside* the loop to see what is happening. Print the variables that appear in the loop condition, so you can see if they are changing or not (if they are not, then the loop is stuck).

Function 1

```
def integrate(lower, upper, nterms):
    # divide the interval into nterms even-sized parts
    delta = (upper - lower) / nterms
    total = 0
    while lower+delta >= upper:
        # compute area from lower to lower + delta
        area = ((1/lower) + (1/(lower + delta))) * delta / 2
    # add to total area
        total = total + area
        lower = lower + delta
    return total
```

Function 2

```
def integrate(lower, upper, nterms):
    # divide the interval into nterms even-sized parts
    delta = (upper - lower) / nterms
    total = 0
    while lower < upper:
        # compute area from lower to lower + delta
        area = ((1/lower) + (1/(lower + delta))) * delta / 2
    # add to total area
        total = total + area
        delta = (upper - lower) / nterms
        lower = lower + delta
    return total
```

Sequence types

We have already seen a number of times that all values in python have a *type*, such as `int`, `float`, `str`, etc. To determine the type of a value we can use the function `type(_some expression_)`. python has three built-in sequence types: lists (type `list`), strings (type `str`) and tuples (type `tuple`). These sequence types are used to represent different kinds of ordered collections. In this lab, we will only use lists, and we will only see a few of the many things that can be done with them. We will return to lists again after the break to examine them in more detail.

To write a list literal, write its elements, separated by commas, in a pair of square brackets:

```
In [1]: x = [1, 2, 3, 4, 5, 6]

In [2]: type(x)
Out [2]: ...
```

The elements that you write can be expressions. These are evaluated, and the resulting values become the elements of the list:

```
In [3]: x = [2, 2 + 1, 2 * 2, 2 + 3]

In [2]: x
Out [2]: ...
```

Indexing sequences

Every element in a sequence has an *index* (position). The first element is at index 0. The *length* of a sequence is the number of elements in the sequence. The index of the last element is the length minus one. The built-in function `len` returns the length of any sequence.

Indexing a sequence selects a single element from the sequence (for example, a character if the sequence is a string). python also allows indexing sequences from the end, using negative indices. That is, `-1` also refers to the last element in the sequence, and `-len(seq)` refers to the first.

Exercise 2

Execute the following in the python shell. For each expression, try to work out what the output will be before you evaluate the expression.

```
In [1]: my_list =  [1, 2, 3, 4, 5, 6]

In [2]: my_list[1]
Out [2]: ...

In [3]: my_list[4]
Out [3]: ...

In [4]: my_list[-1]
Out [4]: ...

In [5]: L = len(my_list)

In [6]: my_list[L - 1]
Out [6]: ...

In [7]: my_list[1 - L]
Out [7]: ...
```

They should all run without error. Is the result of each expression what you expected?

Iteration over sequences

python has two kinds of loop statements: the `while` loop, which repeatedly executes a suite as long as a condition is true, and the `for` loop, which executes a suite once for every element of a sequence. (To be precise, the `for` loop works not only on sequences but on any type that is *iterable*. All sequences are iterable, but later in the course we will see examples of types that are iterable but not sequences.)

Both kinds of loop can be used to iterate over a sequence. Which one is most appropriate to implement some function depends on what the function needs to do with the sequence. The `for` loop is simpler to use, but only allows you to look at one element at a time. The `while` loop is more complex to use (you must initialise and update an index variable, and specify the loop condition correctly) but allows you greater flexibility; for example, you can skip elements in the sequence (increment the index by more than one) or look at elements in more than one position in each iteration.

In the lectures so far, we have only used `while` loops, and they are sufficient to solve all the problems in this lab. We will introduce the `for` loop next week. However, if you want to try using `for` loops, their syntax and execution is described in the text books (Downey: Section "Traversal with a for loop" in Chapter 8; Punch & Enbody: Sections 2.1.4 and 2.2.13).

Exercise 3

The following function takes one argument, which should be sequence of numbers, and computes the average of the numbers in the list:

```
def average(numbers):
    total = 0
    index = 0
    while index < len(numbers):
        total = total + numbers[index]
        index = index + 1
    return total / len(numbers)
```

Test the function with some example inputs. For example

```
In [1]: average([1, 2, 3, 4, 5])
Out [1]: ...

In [2]: average([3, 4, 3, 1])
Out [2]: ...
```

Note that the value returned is always a floating point number (type `float`) even if the average happens to be an even integer.

(a) python has a few built-in functions that work on sequences (of any type): `min(seq)`, `max(seq)` and `sum(seq)` all do what you would expect them to. (Note, however, that `sum` only works on sequences that

contain only numbers.) The function `sorted` returns a list with the elements of the argument sequence in sorted order.

Write a new version of the averaging function that uses `sum`.

(b) Write a function `most_average(numbers)` which finds and returns the number in the input that is *closest* to the average of the numbers. (You can assume that the argument is a sequence of numbers.) By closest, we mean the one that has the smallest absolute difference from the average. You can use the built-in function `abs` to find the absolute value of a difference. For example, `most_average([1, 2, 3, 4, 5])` should return `3` (the average of the numbers in the list is `3.0`, and `3` is clearly closest to this). `most_average([3, 4, 3, 1])` should also return `3` (the average is `2.75`, and `3` is closer to `2.75` than is any other number in the list).

You can use the function above, or the one you just wrote, to compute the average.

Exercise 3(c)

Write a function `count_negative(numbers)` that takes as argument a sequence of numbers and returns number of negative numbers in the sequence. For example, `count_negative([-2, -1, 0, 1, 2])` and `count_negative([2, -2, 1, -1, 0])` should both return `2`, since there are two negative numbers in each argument list. `count_negative([5, 0, 9])` should return `0`, as there are no negative numbers in the input list.

Also test your function on an empty list (that is, a list with no elements). An empty list can be created with the expression `[]` or `list()`. Does your function work?

Exercise 4

A sequence of numbers is said to be (non-strictly) *increasing* if each element is less than or equal to the next element in the sequence. For example `[1, 5, 9]` and `[3, 3, 4]` are both increasing, but `[3, 4, 2]` is not.

Here are two function that are meant to return `True` if the argument sequence is increasing, and `False` otherwise. However, both functions have bugs.

Function 1

```
def is_increasing(seq):
    i = 0
    while i < len(seq):
       if seq[i + 1] < seq[i]:
           return False
       i = i + 1
    return True
```

Function 2

```
def is_increasing(seq):
    i = len(seq) - 1
    while i >= 0:
       if seq[i] < seq[i - 1]:
           return False
       i = i - 1
    return True
```

(a) For each of the two functions, find, if possible, an example of an argument sequence that cause a runtime error due to a list index being out of range. (This may or may not be possible for both functions.)

(b) For each of the two functions, find, if possible, examples of arguments that do not cause a runtime error, but makes the function return the wrong value. (This may or may not be possible for both functions.)

(c) Write a correct function that determines if an argument sequence is increasing.

List comprehension and operations (optional)

python has a mechanism for writing compact expressions that create lists, called *list comprehension*. The general form of a comprehension is

[element_exp for varname in iterable_exp]

iterable_exp should be an expression that evaluates to an iterable type (for example, a sequence), and the comprehension creates a list whose elements are the result of evaluating element_exp for each element in the iterable value. The variable varname is assigned each value from the iterable in turn, and can be used in the element_exp. For example,

```
[ 2 ** k for k in [0, 1, 2, 3, 4] ]
```

will create the list with elements `2 ** 0`, `2 ** 1`, etc, up to `2 ** 4`, i.e, the list `[ 1, 2, 4, 8, 16 ]`.

The built-in function `range(n)` returns an iterable value whose elements are the integers `0`, `1`, etc, up to `n-1`. Thus, the comprehension above could also be written

```
[ 2 ** k for k in range(5) ]
```

You can read more about list comprehensions in Section "List comprehensions" of Chapter 19 in Downey's book, or Section 7.10 in Punch & Enbody's book.

python also allows two operators to applied to lists: `+` and `*`. `+` applied to lists means concatenation. That is, if `A` and `B` are lists, then `A + B` is a list whose elements are all those in `A` followed by all those in `B`. For example,

```
In [1]: [1, 2, 3] + [7, 8]
Out [1]: [1, 2, 3, 7, 8]
```

`*` applied to a list and an integer creates repetition of the elements of the list that many times. For example:

```
In [2]: [1, 2, 3] * 2
Out [2]: [1, 2, 3, 1, 2, 3]
```

Exercise 5 (optional)

Try using list comprehensions, the `range` function and the two list operations `+` and `*` to write compact expressions to create the following lists:

- A list containing N elements that all have the same value, e.g. all are 1.
- A list of integers that counts up from -N to N, in steps of K. For example, for N = 2 and K = 1, the list should be `[-2, -1, 0, 1, 2]`.
- A list of N integers whose value is the same as their index (position in the list) plus 1, that is, `mylist[0] == 1`, `mylist[1] == 2`, `mylist[2] == 3`, etc.

Programming problems

Note: These are more substantial programming problems. We do not expect that everyone will finish all of them during the lab time. If you do not have time to finish them during the lab, you can continue working on them later (at home, in the CSIT labs after teaching hours, or on one of the computers available in the university libraries or other teaching spaces), or come back to them later in the course.

Closest matches

(a) Write two functions, `smallest_greater(seq, value)` and `greatest_smaller(seq, value)`, that take as argument a sequence and a value, and find the smallest element in the sequence that is greater than or equal to the given value, and the greatest element in the sequence that is smaller than or equal to the given value, respectively.

For example, if the sequence is '[3, 1, 13, 5, 9] and the target value is '6', the smallest greater element is 9 and the greatest smaller element is 5.

- You can assume that all elements in the sequence are of the same type as the target value (that is, if the sequence is a list of numbers, then the target value is a number).
- You should *not* assume that the elements of the sequence are in any particular order.
- You should only use operations on the sequence that are valid for all sequence types.
- What happens in your functions if the target value is equal to one of the elements in the sequence?
- What happens in your functions if the target value is smaller or greater than all elements in the sequence?

(b) Same as above, but assume the elements in the sequence are sorted in increasing order; can you find an algorithm that is more efficient in this case?

Counting duplicates

If the same value appears more than once in a sequence, we say that all copies of it except the first are *duplicates*. For example, in `[-1, 2, 4, 2, 0, 4]`, the second 2 and second 4 are duplicates.

Write a function `count_duplicates(seq)` that takes as argument a sequence and returns the number of duplicate elements (for example, it should return 2 for the sequence above). Your function should work on any sequence type (for example, both lists and strings), so use only operations that are common to all sequence types (such as indexing and checking the length). For the purpose of deciding if an element is a duplicate, use standard equality, that is, the `==` operator.

Putting stuff in bins

A *histogram* is way of summarising (1-dimensional) data that is often used in descriptive statistics. Given a sequence of values, the range of values (from smallest to greatest) is divided into a number of sections (called "*bins*") and the number of values that fall into each bin is counted. For example, if the sequence is `[2.09, 0.5, 3.48, 1.44, 5.2, 2.86, 2.62, 6.31]`, and we make three bins by placing the dividing lines at 2 and 4, the resulting counts (that is, the histogram) will be the sequence `2, 4, 2`, because there are 2 elements less than 2, 4 elements between 2 and 4, and 2 elements > 4.

(a) Write a function `count_in_bin(values, lower, upper)` that takes as argument a sequence of numbers and two values that define the lower and upper sides of a bin, and counts the number of elements in the sequence that fall into this bin. You should treat the bin interval as open on the lower end and closed on the upper end; that is, use a strict comparison `lower < element` for the lower end and a non-strict comparison `element <= upper` for the upper end.

(b) Write a function `histogram(values, dividers)` that takes as argument a sequence of values and a sequence of bin dividers, and returns the histogram as a sequence of a suitable type (say, a list) with the counts in each bin. The number of bins is the number of dividers + 1; the first bin has no lower limit and the last bin has no upper limit. As in (a), elements that are equal to one of the dividers are counted in the bin below.

For example, suppose the sequence of values is the numbers 1,..,10 and the bin dividers are `[2, 5, 7]`; the histogram should be `[2, 3, 2, 3]`.

To implement this function, you may need to grow the size of the list that represents the histogram. You can do this with the list concatenation operator: if `mylist` is a list (which can be empty) and `x` is a value (number) that you want to add to it, then the assignment `mylist = mylist + [x]` adds `x` to the end of `mylist`. You can also use list comprehension, as described above.

To test your function, you can create arrays of random values using NumPy's random module:

```
In [1]: import numpy.random as rnd
In [2]: values = rnd.normal(0, 1, 50)
```

This creates an array of 50 numbers drawn according to the normal distribution with mean 0 and standard deviation 1. The following creates 10 evenly sized bins covering the range of values:

```
In [1]: import numpy as np
In [2]: range = np.max(values) - np.min(values)
In [3]: dividers = (np.arange(1, 10) * (range / 10)) + np.min(values)
```

As you increase the size of the value array, you should find that the histogram becomes more symmetrical and more even.

You can also test your function by comparing it with the histogram function provided by NumPy (see `help(numpy.histogram)`).